МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА КОМПИЛЯТОРА СРЕДСТВАМИ .NET

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Чауенова Камиля Рашидовича

Научный руководитель
зав. каф. к. ф.-м. н. _____ С. В. Миронов

Заведующий кафедрой
к. ф.-м. н. _____ С. В. Миронов

Саратов 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы. Компилятор – компьютерная программа, переводящая компьютерный код, написанный на высокоуровневом языке программирования, в низкоуровневый машинный язык.

Компилятор для относительно простого языка, написанный одним человеком, может представлять собой единую, монолитную часть программного обеспечения. Однако по мере усложнения исходного языка разработка может быть разделена на несколько взаимосвязанных этапов.

Современные вызовы разработки программного обеспечения, такие как рост сложности систем, требования к безопасности и производительности, обуславливают необходимость создания специализированных инструментов.

Цель данной работы — разработать компилятор высокоуровневого языка программирования со статической типизацией. В рамках достижения этой цели ставятся задачи:

- провести анализ техник лексического и синтаксического анализа и выберем наилучший инструмент;
- определить способы описания семантики языка программирования и построения семантической структуры;
- рассмотреть способы генерации кода перед трансляцией в исполняемый код и способы его оптимизации;
- провести консолидацию процесса компиляции в единое целое.

Практическая значимость бакалаврской работы заключается в создании методики собственного компилятора языка программирования с использованием современных средств создания трансляторов, методов описания формальных языков и построения семантики.

Структура и объём работы. Бакалаврская работа состоит из введения, 2 разделов, заключения, списка использованных источников и 4 приложений. Общий объем работы – 71 страниц, из них 43 страниц – основное содержание список использованных источников информации – 20 наименований.

1 КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

Процесс обработки компилятором можно разделить на три уровня: поверхностный, средний и внутренний уровень.

Проект будет включать в себя все этапы обработки кода, написанного в рамках определенной грамматики и правил. Созданный язык программирования будет поддерживать статическую типизацию, управляющие конструкции и объектно-ориентированное программирование. Выполнение кода будет осуществляться с помощью JIT-компиляции.

Практическая часть работы включает следующие этапы:

1. Проектирование лексики, синтаксиса и семантики: определение ключевых конструкций языка, системы типов. Разработка грамматики с использованием формальных методов (BNF-нотация).
2. Реализация интерпретатора. Построение конвейера обработки кода: лексический и синтаксический анализ (на базе инструмента ANTLR4), семантический анализ (проверка типов), генерация промежуточного кода (LLVM IR).
3. Написание тестовых сценариев для проверки корректности типизации, обработки ошибок и производительности.

1.1 Лексический и синтаксический анализ

Лексический анализ представляет собой важнейший этап в процессе разработки компилятора. Его основная задача заключается в преобразовании входного потока символов в последовательность лексем (токенов) — логических единиц, имеющих определённое значение для последующих этапов компиляции.

Синтаксический анализ является следующим этапом обработки входной строки на соответствие правилам формальной грамматики.

Опишем лексику данного языка программирования. Токены, составляющие код, можно разделить на следующие категории:

— Ключевые слова. Зарезервированные слова для управления логикой программы:
  – для управления потоком управления программы: условные операторы, операторы цикла;
  – для определения функций;

  - для определения классов и механизмов ООП;
  - сигнатуры примитивных типов.
- Литералы;
  - целые числа;
  - вещественные числа;
  - строки;
  - логические значения.
- идентификаторы будут представлены последовательностью латинских букв;
- разделители;
- операторы;
- комментарии – строки, которые не будут обрабатываться семантическим анализатором и интерпретатором.

Литералы были описаны в виде регулярных выражений.

В работе были проанализированы разные способы разбора входной строки на основе техник лексического и синтаксического анализа. В рамках поставленной задачи был выбран ANTLR4 в качестве генератора лексера и парсера, использующих алгоритм ALL(*) разбора ввиду некоторых преимуществ перед другими генераторами.

Синтаксис языка программирования был описан с помощью EBNF-нотации. Нотация была использована для генерации лексера и парсера языка программирования с помощью ANTLR4.

Дерево разбора сгенерированное ANTLR4 отражает собой синтаксис строки. В свою очередь ANTLR4 предлагает два способа обхода дерева с использованием паттерна Посетитель и Слушатель. Оба подхода позволяют реализовать семантический анализ и преобразование кода, но различаются в способе управления обходом дерева и организации логики обработки узлов.

Паттерн "Слушатель"основан на событийной модели: ANTLR4 генерирует классы, которые автоматически обходят дерево разбора в порядке depth-first (поиск в глубину), инициируя вызовы предопределённых методов при входе в узел (enterNode) и при выходе из него (exitNode).

В отличие от "Слушателя паттерн "Посетитель"требует явного вызова методов обхода через метод visit(). Это обеспечивает гибкий контроль над порядком посещения узлов и позволяет возвращать значения из каждого узла.

Так как при семантическом анализе и генерации промежуточного кода

необходимо гибкое управление обхода дерева, сгенерируем класс посетителя с помощью ANTLR4.

1.2 Семантический анализ

Семантический анализ в компиляторе связывает определения переменных с их назначением, проверяет каждое выражение и его тип, проверяет поток управления или преобразует программу в форму удобную для дальнейшего анализа.

Создадим символьную таблицу для хранения информации о переменных и их типах. В качестве основы реализуем вспомогательный класс Symbol. Для эффективного хранения строк воспользуемся механизмом интернирования строк в .NET CLR.

В дальнейшем будем использовать этот класс для хранения названий переменных. Сами переменные и информация об их типах будут содержаться в классе *Binder*. Для реализации входа и выхода из области видимости добавим дополнительное поле для хранения ссылки на предыдущее значение переменной.

Определим класс символьной таблицы. Таблица должна поддерживать сохранение области видимости, хранение информации о текущем классе и функции.

Таблица реализована на основе хеш-таблицы со следующими особенностями:

При добавлении связи $x \to b$ (через $table.put(x, b)$):

1. Ключ *x* хешируется для определения индекса *i*
2. В начало связного списка для *i*-го бакета помещается объект *Binder* с привязкой $x \to b$
3. Существующие привязки $x \to b$ остаются в списке, но становятся невидимыми (перекрываются новой привязкой)
4. Вспомогательный стек фиксирует порядок добавления символов:
5. При добавлении $x \to b$ символ x помещается в стек
6. Операция *beginScope* добавляет в стек специальный маркер
7. При выполнении *endScope*: из стека последовательно извлекаются элементы до ближайшего маркера и для каждого извлеченного символа удаляется соответствующая привязка.

Маркеры в стеке содержат данные о переменных и функциях определенных в

области видимости. Дополнительно добавим возможность извлекать текущий обрабатываемый класс и функцию.

Семантический анализ будет происходить в виде проверки типов и последующего преобразования в абстрактное синтаксическое дерево. Данная структура позволяет провести ряд оптимизаций перед дальнейшей генерацией промежуточного кода, в частности, распространение и свёртку констант. В отличие от дерева, сгенерированного ANTLR4, в нем не будет вершин, не используемых во время генерации промежуточного кода или проверки типов.

Правила вывода проверки типов позволяют формализовать корректность семантики программы. Сами правила были записаны в виде формулы:

$$\frac{\vdash \text{Гипотеза}}{\vdash \text{Вывод}}$$

Проверка типов осуществлялся в соответствии с заданными правилами вывода. При идентификации ошибки пользователь получает сообщение о ней в консоли. В некоторых случаях может производиться неявное приведение типов.

Кроме правил вывода проверки типов существует операционная семантика. С помощью неё можно определить, какое значение будет порождаться в данном контексте. Контекст включает в себя среду, хранилище и экземпляр класса.

Правила вывода операционной семантики выглядят следующим образом:

$$\frac{\vdots}{th, S, E \vdash e_1 : u, S'}$$

где th – это объект класса, выражение e_1 сводится к объекту u и помещается в хранилище S'.

Правила вывода операционной семантики использовались для последующих фаз компиляции.

Таким образом сгенерируется абстрактное синтаксическое дерево, которое можно использовать для дальнейшего анализа кода и генерации промежуточного кода.

При наследовании и имплементации наследник получает от родителя его методы и поля в том порядке, которые они были определены в самом родителе. В отдельных случаях, методы могут быть переопределены. Отношение наследования является частично-упорядоченным, т.е некоторые отдельные подпо-

следовательности могут следовать друг за другом, но не обязательно, что это справедливо для каждой пары элементов.

Для отслеживания атрибутов и методов был выбран порядок разрешения методов или C3-линеаризация.

Алгоритм C3-линеаризации является подходом, устраняющим неоднозначность при построении иерархии классов, их полей и методов. В ходе работы алгоритма строится последовательность элементов, классов. Данная последовательность является частично-упорядоченным множеством. Это свойство позволяет автоматизировать механизм множественного наследования.

1.3 Генерация промежуточного кода в LLVM

В данном разделе приводится реализация генерации промежуточного кода с использование инфраструктуры создания компиляторов.

Воспользуемся проектом программной архитектуры LLVM для генерации промежуточного представления, оптимизации кода и выполнении кода.

Разработка на .NET будет осуществляться с помощью библиотеки LVVMCSharp.

Множество виртуальных инструкций LLVM охватывают ключевые операторы свойственные машиннозависимым языкам, но без ограничений в виде количества регистров или конвейеров. Перемещение значений в LVVM осуществляется с помощью операций $load$ и $store$. Для выделения памяти в стеке используется команда $allocate$ со строгой спецификацией типа.

Инструкции в LVVM имеют трех-адресное представление, включая арифметические и логические операции: $add, sub, mul, div, rem, not, and, or, xor$ и др. Данные инструкции полиморфны, поэтому возможно совмещать целочисленные и вещественные значения.

Переменные объявляются в SSA форме, что делает возможным провести ряд оптимизаций над кодом. Каждая инструкция, вычисляющая значение, неявно создает виртуальный регистр, хранящий значение. Это позволяет ссылаться на каждую конкретную операцию.

— LLVMModuleRef – основной контейнер для всего сгенерированного кода. Содержит функции, глобальные переменные и другие структуры. Модуль в LLVM - это аналог единицы компиляции (например, файла .c в C).

— LLVMBuilderRef – "строитель"IR-инструкций. Этот объект предоставляет методы для последовательного создания инструкций LLVM IR внутри

базовых блоков функций. Отвечает за правильное размещение инструкций в коде.

— LLVMExecutionEngineRef – исполняющий движок LLVM, который может выполнять сгенерированный код напрямую (JIT-компиляция) или преобразовывать его в машинный код.
— LLVMOpaquePassBuilderOptions – Указатель на объект с настройками для оптимизационных проходов LLVM. Определяет, какие оптимизации будут применяться к сгенерированному коду.
— Ранее определенная таблица символов.

Литералы преобразуются в LLVM-константы:

— Логические — в однобитные целочисленные (i1)
— Числовые — в соответствующие типы (i32, double и др.)

Арифметические операции реализуются через LLVM-инструкции:

— BuildFAdd, BuildFSub — для арифметики
— BuildFCmp — для сравнений с конвертацией в логический тип

Работа с переменными:

— Объявление — выделение памяти в стеке (BuildAlloca)
— Присваивание — сохранение значения (BuildStore)
— Таблица символов хранит указатели на выделенную память

Управление потоком управления было реализовано следующим образом.

— Основано на системе базовых блоков.
— Условные переходы:
  – Генерация блоков для условия, ветвей и слияния
  – Инструкция BuildCondBr для ветвления
  – Позиционирование генератора через PositionAtEnd
— Рекурсивный обход AST для заполнения блоков

Создание прототипа функции в LLVM заключалось в определении возвращаемого типа и типов аргументов.

Внутренняя структура функции:

— Входной блок и блок возврата.
— Размещение параметров в стеке.
— Выделение памяти под возвращаемое значение.
— Вызовы через BuildCall

Используя функционал выделения памяти в куче и гибкое управление

памятью и указателями объектно-ориентированное программирование может быть реализовано с помощью LLVM следующим образом:

- Классы как LLVM-структуры:
  - Определение через StructSetBody
  - C3-линеаризация полей
- Инстанцирование объектов:
  - Выделение памяти через malloc
  - Приведение типа через BuildBitCast
- Методы:
  - Первый параметр — неявный this
  - Обращение к полям через указатели

1.4 Оптимизация промежуточного представления

Перед выполнением сгенерированного кода можно воспользоваться оптимизаторами предоставляемые библиотекой LVVM. В LLVM проходы (passes) — это оптимизации или преобразования, которые изменяют промежуточное представление программы для улучшения её производительности, уменьшения размера кода или подготовки к дальнейшей обработке.

Во время генерации промежуточного кода генерируется граф потока управления и граф потока данных, позволяющих внедрить оптимизации в код. Добавим следующие проходы:

- simplifycfg – упрощает граф потока управления, удаляя лишние ветвления и блоки.
- mem2reg – преобразует операции с памятью в регистровые переменные.
- gvn – устраняет избыточные вычисления и загрузки, находя повторяющиеся значения в пределах всей функции.

CFG в LLVM представляет из себя вершины в виде блоков, соединенных инструкциями переходами. Принцип simplifycfg заключается в оптимизации «глазок» упрощающих вид графа.

Удаление недостижимого кода является ключевой операции в упрощении вида графа. Любые «висящие» блоки в графе будут удалены, как недостижимые. Удаляются как единичные блоки, так и группы недостижимые для главного потока управления.

mem2reg выполняет специфичную для LLVM IR оптимизацию с использованием регистров, которые являются бесконечными в промежуточном пред-

ставлении. После данного прохода alloca, load, store инструкции заменяются на регистры и ϕ-функции с сохранением SSA-формы.

Используется граф потока данных, отслеживающий отношение определения и использования между блоками. Зная, в каком блоке произошло определение и использование переменной можно удалить alloca, load и store инструкции, подставив регистр.

Gloval Value Numbering устраняет дублирующие вычисления и реализует распространение констант. Одним и тем же значениям в LLVM IR присваивается одинаковые номера. Множество инструкций с одинаковым номером заменяется на одно и то же представление. В случае арифметических или логических команд генерируется уникальный номер, а вызовы функции учитывают параметры. Таким образом, строится отображение инструкций в целочисленные значения.

Алгоритм обходит блоки в обратном порядке, находя и изменяя дублирующие инструкции по их номерам. Это ведет к удалению общих подвыражений, оптимизируя вычисления.

Подсчет значений выполняется в несколько проходов, фиксируя изменения номеров в разных участках кода. При достижении неизменяемости номеров, алгоритм прекращает свою работу. Как и в предыдущем случае волатильные операции загрузки будут проигнорированы.

ЗАКЛЮЧЕНИЕ

Компиляторы – фундаментальная часть программного обеспечения, позволяющая создать слой абстракции между вычислительной машиной и программистом. Код, генерируемый компилятором может использоваться для решения самых разных задач повседневной жизни и не только. В данной работе была предпринята попытка изучить процесс создания компилятора языка программирования со статической типизацией в рамках определенных правил лексики, синтаксиса и семантики.

Удалось достичь следующих результатов:

- Был сгенерирован лексический и синтаксический анализатор, разбирающий входную строку в последовательность токенов и синтаксических конструкций.
- Определены правила вывода семантического анализа. Реализована проверка типов и построение абстрактно-синтаксического дерева.
- Написан генератор промежуточного кода с помощью инструмента LLVM, реализующего функционал языка программирования с использование JIT-компиляции.
- Протестирован функционал языка программирования.

В заключение, процесс разработки компилятора – достаточно сложный и трудоемкий процесс, охватывающий различные аспекты разработки программного обеспечения, однако правильная реализация гарантирует, что программист сможет использовать вычислительные мощности наиболее эффективным способом.